

CS190C Lec6

Training and Autoregressive Decoding

Overview

- Model Training: Macro System Architecture
- Training modules
 - Data Sampling Module
 - AdamW Optimizer
 - Training Auxiliary Modules
 - Checkpoints
- Training and Decoding
 - Start of Training: Input Parameter Decoding
 - Complete script
 - Autoregressive Decoding

PART1 - Model Training: Macro System Architecture

Overview of Model Training Process

- Determine the total number of iterative optimization steps. For each step:
 - Sample input text and standard output.
 - Forward.
 - Calculate loss function value.
 - Calculate gradients and clipping.
 - Schedule the learning rate.
 - Update parameters.
- It is recommended to output real-time status during the process, such as the current loss function value.

Modules to Implement

- Input & Standard output data ⇒ Data Sampling Module
- Loss calculation ⇒ Cross-Entropy Loss Module
- Calculate and process Gradients ⇒ Gradient Clipping Module
- Parameter Optimization ⇒ Learning Rate Scheduler Module, AdamW Optimizer Module
- Checkpoint Module, Pre-decoding Module

PART2: Data Sampling Module

Memory Issues with Very Large Corpus

- The corpus will initially encoded to a long string of numbers using BPE.
- To sample a part of it as a "sentence", a naive method is to read the whole long string into memory, and sample hundreds of numbers.
- If the corpus is extremely large, reading all of them will bring disaster to memory.



Memory Issues with Very Large Corpus

- Is there a solution that reads only specific parts needed from disk to memory?
- `numpy.memmap` : A list that interacts directly with disk space for reading and writing, and highly similar to a regular list in other aspects.
- We can implement `class Memmap_Manager`, containing `def save_as_memmap` and `def load_by_range`, for disk-oriented read/write operations.
- The core idea is: Store the whole long string in the disk, only read a small part of it to memory according to our needs.

Ideas

We can realize this idea roughly.....

- Suppose the whole long string contains 500,000,000 numbers.
- We can cut it into small chunks, each chunk's length= 500,000 .
- Store these 10000 chunks into the disk.
- Suppose we need to read 2000th-8000th number, what chunks should we load to memory? 490,000th-1,050,000th ?
- These two phases correspond to def save_as_memmap and def load_by_range .

1. `def save_as_memmap`

- Use `BPE_Tokenizer`'s `encode_iterable`, reading only a single number each time.
- Continuously read numbers into a buffer list.
- When the buffer reaches a certain size (e.g., 500,000 numbers), write it as a whole block to disk.
- Clear the buffer and continue reading and writing blocks until the corpus is finished.
- Count the total number of integer codes in the corpus during the process.

That is: **saving as encoding**.

1. def save_as_memmap

```
def save_as_memmap(self):
    tokenizer=BPE_Tokenizer.from_files(self.vocab_path,self.merge_path,self.special_tokens)
    buffer=[]
    chunk_num=0
    length=0

    with open(self.corpus_path) as f:
        encoder=tokenizer.encode_iterable(f)
        for id in encoder:
            length+=1
            buffer.append(id)
            if len(buffer)>=self.chunk_size:
                self.save_by_chunks(buffer,self.chunk_size,chunk_num)
                chunk_num+=1
                buffer=[]
        if len(buffer)>0:
            self.save_by_chunks(buffer,len(buffer),chunk_num)
            buffer=[]

    print(f"length of corpus in tokens:{length}")
```

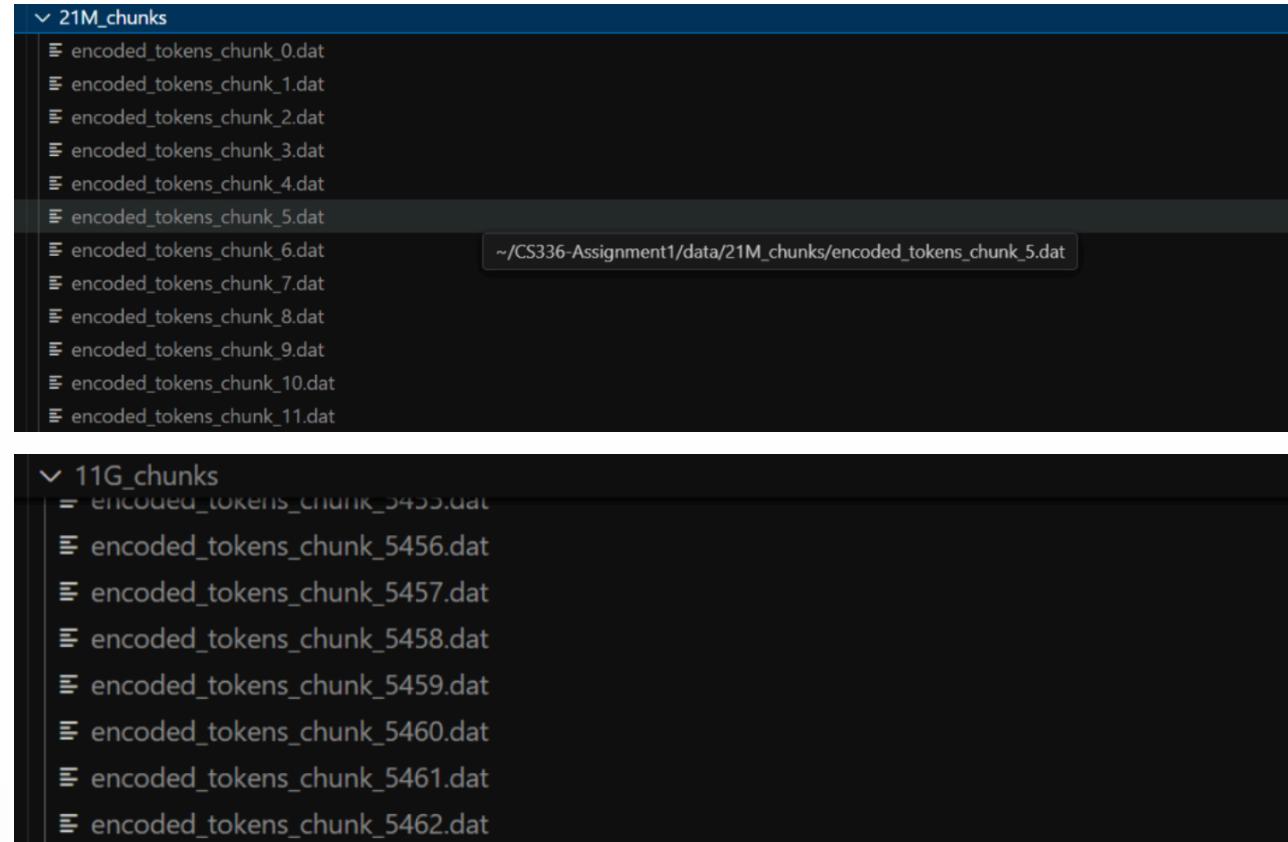
1. def save_as_memmap

```
def save_by_chunks(self, token_ids, buffer_len, chunk_num):
    fname = "/home/kuangph/CS336-Assignment1/data/" + self.corpus_size + f"_chunks/encoded_tokens_chunk_{chunk_num}.dat"
    dtype = np.int32
    shape = (buffer_len,)
    memmap_arr = np.memmap(fname, dtype=dtype, mode="w+", shape=shape)
    memmap_arr[:] = token_ids[:]
    memmap_arr.flush()
```

For np.memmap:

- Define array type and shape in advance.
- Operate like a regular array, still stored in memory.
- Explicit flush operation writes to disk, clearing previous memory automatically.

1. def save_as_memmap



Attempting to load such a large complete list into memory is catastrophic!

2. `def load_by_range`

Requirement: Need to read all elements in the range `[start_idx, end_idx)` of the encoded list.

- Calculate: Which chunks covered?
- Load only corresponding intervals into memory.
- Operations are consistent with regular lists, except declaring the `np.memmap` type.

2. def load_by_range

memmap_arr[idx_in_start:idx_in_end] : Loads this interval from disk into memory.

```
def load_by_range(self,start_idx,end_idx):
    chunk_size=self.chunk_size
    start_chunk=start_idx//chunk_size
    end_chunk=end_idx//chunk_size
    idx_in_start=start_idx%chunk_size
    idx_in_end=end_idx%chunk_size

    token_ids=[]
    for chunk in range(start_chunk,end_chunk+1):
        fname=f"/home/kuangph/CS336-Assignment1/data/{self.corpus_size}Chunks/encoded_tokens_chunk_{chunk}.dat"
        dtype=np.int32
        memmap_arr=np.memmap(fname,dtype=dtype,mode="r")
        if start_chunk==end_chunk:
            token_ids.extend(memmap_arr[idx_in_start:idx_in_end])
        else:
            if chunk==start_chunk:
                token_ids.extend(memmap_arr[idx_in_start:])
            elif chunk>start_chunk and chunk<end_chunk:
                token_ids.extend(memmap_arr[:])
            else:
                token_ids.extend(memmap_arr[:idx_in_end])
    return token_ids
```

3. Formal Sampling Operation: `class Batch_By_Memmap`

Problem1: How to sample input data?

Assuming a corpus length of 10 and a single sampling sequence length of 4:

- Dense sampling: 1234; 2345; 3456; ...
 - Too much data. If the total number of training steps is small (i.e., 3 samples), the second half of the corpus might not be covered.
- Sequential sampling: 1234; 5678; ...
 - Some data is missing. E.g., cannot get 3456 as training input.
- Compromise: Random sampling
 - Randomly pick a start point (ensuring it's ≤ 7), then continuously sample 4 characters from the start point.
 - Usually sample `batch_size` sequences simultaneously, so randomly pick `batch_size` start points to sample.

3. Formal Sampling Operation: `class Batch_By_Memmap`

Problem2: What is the standard output?

- Review what LLM done: each vector is finally linearly projected to `vocab_size` scale weights to predict the next position's word (as the diagram below).
- So the standard output of position- i should be the $i + 1$ -th word in the corpus !
- That is: if the input sequence is 1234, the standard output should be 2345.



3. Formal Sampling Operation: `class Batch_By_Memmap`

```
class Batch_By_Memmap:
    def __init__(self,memmep_manager:Memmap_Manager):
        self.memmap_manager=memmep_manager

    def get_batch(self,bsz,seq_len,dataset_length,device=None):
        max_start_idx=dataset_length-seq_len
        start_indices=np.random.randint(0,max_start_idx,bsz)

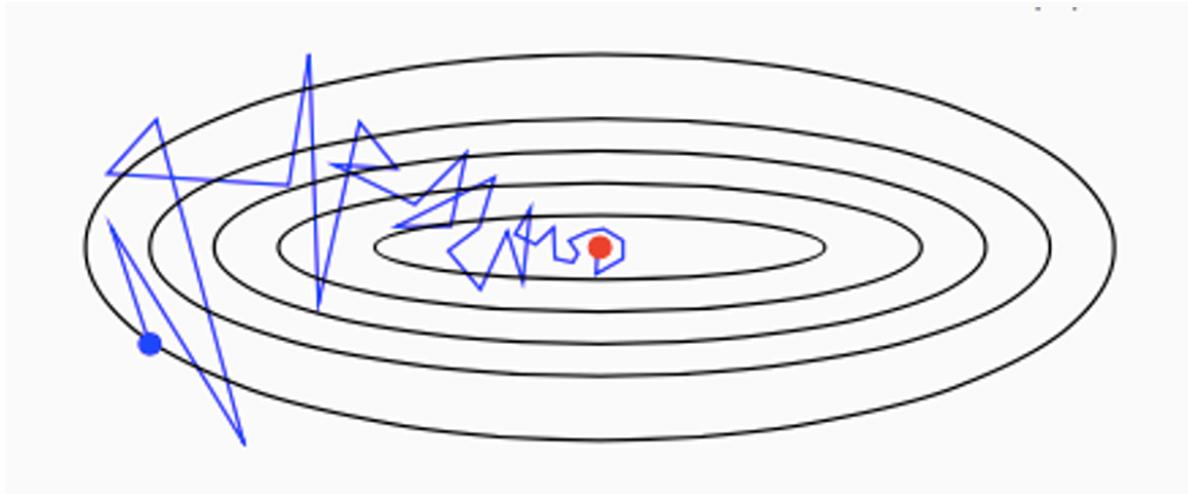
        x=np.array([self.memmap_manager.load_by_range(i,i+seq_len) for i in start_indices],dtype=np.int64)
        y=np.array([self.memmap_manager.load_by_range(i+1,i+seq_len+1) for i in start_indices],dtype=np.int64)

        x=torch.tensor(x,dtype=torch.long,device=device)
        y=torch.tensor(y,dtype=torch.long,device=device)
        return (x,y)
```

PART3: AdamW Optimizer

1. From SGD to AdamW

$$\text{SGD: } \theta = \theta - \alpha \nabla L(\theta)$$



1. From SGD to AdamW

What's the defect of SGD?

- Zigzagging because the gradient may not stable.
- Share the same learning rate for updating of all parameters
 - Some parameter, such as word embedding of `the`, may update repeatedly and significantly, leading to possible oscillation.
 - Some parameter, such as word embedding of `pneumoconiosis`, may rarely update, leading to little optimization.

1. From SGD to AdamW

What we do in AdamW ?

- Use first moment m (we called "momentum") to record historical gradients information.
- Use second moment v to record historical gradients fluctuation information.
- Use first and second moment to adjust parameter update process.

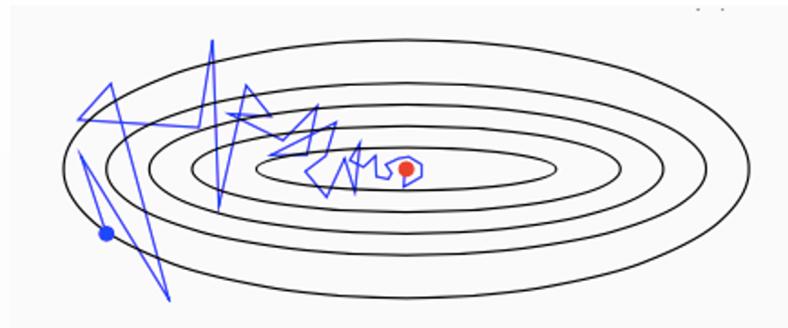
1. From SGD to AdamW

Let gradient of step t be g_t :

$$m_t = \beta_1(m_{t-1} - g_t) + g_t$$

It means: $m_t = (1 - \beta_1) \sum_{\tau=1}^t \beta_1^{t-\tau} g_\tau$, it is the exponential weighting of historical gradients.

We use m_t as the "gradient" of step t instead of g_t , reducing zigzagging of gradient. m_t now can hardly fluctuate so serious.



1. From SGD to AdamW

$$v_t = \beta_2(v_{t-1} - g_t^2) + g_t^2$$

It means: $v_t = (1 - \beta_2) \sum_{\tau=1}^t \beta_2^{t-\tau} g_\tau^2$, it is the exponential weighting of historical square of gradients.

And also, it can understand as: Variance given mean=0 and exponential weights . But what if mean $\neq 0$?

Suppose a sequence of gradients with the same mean value: 2,2,2,2 3,1,3,1 , we find 3,1,3,1 still leads to larger value of v . So it can reflect the fluctuation of historical gradients.

1. From SGD to AdamW

So how do we use these two information to adjust $\theta_t = \theta_{t-1} - \alpha g_t$?

- For first moment, change g_t to m_t
- How to make use of secomd moment?
 - If v is large: It means the gradient of parameter fluctuate seriously, such as word embedding of high frequency words. The update step should be smaller.
 - If v is small: It means the parameter rarely update, for example, the history gradient are $0, 0, 0, 1, 0, 0$. The update step should be larger.

$$\theta_t = \theta_{t-1} - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon}$$

1. From SGD to AdamW

$$\theta_t = \theta_{t-1} - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon}$$

Is there any hidden problem of it?

When t is small.....

- $m_1 = (1 - \beta_1)g_1, m_2 = \beta_1(1 - \beta_1)g_1 + (1 - \beta_1)g_2.....$
- Suppose all $|g_i|$ share the similar scale, so $|m_1| = (1 - \beta_1)|g|, |m_2| = (1 - \beta_1^2)|g|, \dots, |m_t| = (1 - \beta_1^t)|g| \Rightarrow$ The scale is distortion comparing with $|g|$
- The same with $|v_t|: |v_t| = \sqrt{1 - \beta_2^t}|g|$

1. From SGD to AdamW

Fix the scale distortion: $m_t \Rightarrow \frac{m_t}{1-\beta_1^t}$, $v_t \Rightarrow \frac{v_t}{\sqrt{1-\beta_2^t}}$

That is:

$$\theta_t = \theta_{t-1} - \alpha \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \frac{m_t}{\sqrt{v_t} + \epsilon}$$

We can implement it in engineering:

$$\alpha_t = \alpha \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}$$

$$\theta_t = \theta_{t-1} - \alpha_t \frac{m_t}{\sqrt{v_t} + \epsilon}$$

1. From SGD to AdamW

So far: It is the adjustment of SGD without regularization. What if we consider $L2$ -regularization?

$$\begin{aligned} L_{total}(\theta) &= L_{task}(\theta) + \frac{\lambda}{2} \|\theta\|_2^2 \\ \Rightarrow \theta &= \theta - \alpha \nabla L_{task}(\theta) - \alpha \lambda \theta \end{aligned}$$

The same to AdamW :

$$\begin{aligned} \theta_t &= \theta_{t-1} - \alpha \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t} \frac{m_t}{\sqrt{v_t} + \epsilon} - \alpha \lambda \theta \\ &= \theta_t = \theta_{t-1} - \alpha_t \frac{m_t}{\sqrt{v_t} + \epsilon} - \alpha \lambda \theta \end{aligned}$$

1. From SGD to AdamW

What the order of magnitude of scale of updating?

SGD: $|\alpha g_t| = \Theta(\alpha |g|)$

AdamW: $|\alpha_t \frac{m_t}{\sqrt{v_t + \epsilon}}| = \Theta(\alpha_t \frac{\sum_{k=1}^t \beta_1^k |g|_{t-k}}{\sqrt{\sum_{k=1}^t \beta_2^k |g|_{t-k}^2}})$

When the training process enter the middle and late stages,

$$\alpha_t \rightarrow \alpha$$

$|g|$ can be considered as a sequence with mean = 0 and variation = σ^2

$$\Rightarrow |g| \rightarrow \Theta(\sigma)$$

1. From SGD to AdamW

$$\text{AdamW: } |\alpha_t \frac{m_t}{\sqrt{v_t + \epsilon}}| = \Theta(\alpha_t \frac{\sum_{k=1}^t \beta_1^k |g|_{t-k}}{\sqrt{\sum_{k=1}^t \beta_2^k |g|_{t-k}^2}})$$

$$|m_t| = \sum_{k=1}^t \beta_1^k |g|_{t-k} = \Theta(\sigma)$$

$$|\sqrt{v_t}| = \sqrt{\sum_{k=1}^t \beta_2^k |g|_{t-k}^2} = \Theta(\sqrt{\sigma^2}) = \Theta(\sigma)$$

$|\alpha_t \frac{m_t}{\sqrt{v_t + \epsilon}}| = \Theta(\alpha)$, which is only related to base learning rate.

2. `torch.optim.Optimizer` Base Class

Any implementation of a custom optimizer inherits from the `torch.optim.Optimizer` base class.

The Optimizer base class provides two-level management for all model parameters:

- `self.param_groups` : Group parameters. Each group can set different features like learning rate. (First-level management: parameter grouping, sharing states within the group)
- Each `param_groups` contains at least the default key-value pair: "params", corresponding to a list of parameters. Other key-value pairs like "lr" can also be customized.
- For each parameter in the parameter list, various states can also be set, e.g., iteration step `"step"` . (Second-level management: finer-grained states for individual parameters)

2. torch.optim.Optimizer Base Class

```
class AdamW_Optimizer(torch.optim.Optimizer):
    def __init__(self, parameters, lr:float, weight_decay:float, betas, eps:float):
        param_groups=[
            {
                "params":parameters,
                "lr":lr
            }
        ]
        super(AdamW_Optimizer, self).__init__(param_groups, {})
        self.weight_decay=weight_decay
        self.beta1=betas[0]
        self.beta2=betas[1]
        self.eps=eps

    for group in self.param_groups:
        for p in group["params"]:
            self.state[p]={
                "m":torch.zeros_like(p.data),
                "v":torch.zeros_like(p.data),
                "step": torch.tensor(0.0, device=p.device)
            }
```

3. class AdamW_Optimizer

Algorithm 1 AdamW Optimizer

```
init( $\theta$ ) (Initialize learnable parameters)
 $m \leftarrow 0$  (Initial value of the first moment vector; same shape as  $\theta$ )
 $v \leftarrow 0$  (Initial value of the second moment vector; same shape as  $\theta$ )
for  $t = 1, \dots, T$  do
    Sample batch of data  $B_t$ 
     $g \leftarrow \nabla_{\theta} \ell(\theta; B_t)$  (Compute the gradient of the loss at the current time step)
     $m \leftarrow \beta_1 m + (1 - \beta_1)g$  (Update the first moment estimate)
     $v \leftarrow \beta_2 v + (1 - \beta_2)g^2$  (Update the second moment estimate)
     $\alpha_t \leftarrow \alpha \frac{\sqrt{1 - (\beta_2)^t}}{1 - (\beta_1)^t}$  (Compute adjusted  $\alpha$  for iteration  $t$ )
     $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v} + \epsilon}$  (Update the parameters)
     $\theta \leftarrow \theta - \alpha \lambda \theta$  (Apply weight decay)
end for
```

```
def step(self):
    for group in self.param_groups:
        for p in group['params']:
            if p.grad is None:
                continue

            grad=p.grad.data
            state=self.state[p]
            m,v,step=state["m"],state["v"],state["step"]

            if not isinstance(step,torch.Tensor):
                step=torch.tensor(float(step),device=p.device)
                state["step"]=step

            current_lr=group.get("lr")

            m=m*self.beta1+(1-self.beta1)*grad
            v=v*self.beta2+(1-self.beta2)*(grad**2)
            step=step+1

            alpha_t=current_lr*(math.sqrt(1-self.beta2**step)
                                /(1-self.beta1**step))
            p.data=(p.data
                    -alpha_t*(m/(torch.sqrt(v)+self.eps))
                    -current_lr*self.weight_decay*p.data)

            self.state[p]["m"]=m
            self.state[p]["v"]=v
            self.state[p]["step"]=step
```

PART4: Training Auxiliary Modules

1. class Cross_Entropy_Calculator

Review: What LLM done, and what's the meaning of it.

- Transformer output shape is $[bsz, seq_len, vocab_size]$. I.e., there are $(bsz * seq_len)$ output predictions, each prediction is "the possibility score s_i for each word at the next position" ($1 \leq i \leq bsz * seq_len$), same below.
- Perform Softmax to get probability weight distribution p_i of next word.



1. class Cross_Entropy_Calculator

- Take $-\log$ of the probability weight to get $l_i = -\log(p_i)$, the mean of all l_i is the **cross-entropy loss**.
- That is: Loss = Mean $\{-l_i\}$ = Mean $\{-\log p_i\}$ = Mean $\{-\log \text{Softmax}(s_i)\}$



1. class Cross_Entropy_Calculator

Numerical hazard: What if p_i is too small and approximated as 0?

- Suppose token1's 6-word weight scores are 100, -1, 1, 5, 2, 1000, and the standard output is word2, with score -1 and weight 0.
- Calculation result: $\log 0 = \text{NAN}$

Formula derivation:

$$\begin{aligned}\text{logSoftmax}(s_i) &= \log \frac{\exp(s_i)}{\sum \exp s_k} \\ &= \log \frac{\exp(s_i - s_{\max})}{\sum \exp(s_k - s_{\max})} \\ &= (s_i - s_{\max}) - \log(\sum \exp(s_k - s_{\max}))\end{aligned}$$

1. class Cross_Entropy_Calculator

```
class Log_Softmax():
    def __init__(self, dim:int=-1):
        self.dim=dim

    def forward(self, x:torch.Tensor)->torch.Tensor:
        x_max=torch.max(x, dim=self.dim, keepdim=True).values
        x=x-x_max
        x_exp=torch.exp(x)
        x_exp_sum=torch.sum(x_exp, dim=self.dim, keepdim=True)
        return x-torch.log(x_exp_sum)
```

1. class Cross_Entropy_Calculator

```
class Cross_Entropy_Calculator:  
    def __init__(self):  
        self.log_softmax=Log_Softmax(dim=-1)  
  
    def forward(self,inputs:torch.Tensor,targets:torch.Tensor)->torch.Tensor:  
        inputs=inputs.reshape(-1,inputs.shape[-1])#[bsz*seq_len,vocab_size]  
        inputs=-self.log_softmax.forward(inputs)#[bsz*seq_len,vocab_size]  
        targets=targets.reshape(-1)#[bsz*seq_len]  
  
        selected=inputs[torch.arange(inputs.shape[0]),targets]#[bsz*seq_len]  
        loss=torch.mean(selected,dim=0)  
        return loss
```

2. class Gradient_Clipper

Suppose the gradient of parameter is too large?

- SGD: $|\alpha g_t| = \Theta(\alpha |g|)$. It may lead to failure of optimization.
- AdamW: $|\alpha_t \frac{m_t}{\sqrt{v_t + \epsilon}}| = \Theta(\alpha)$. It seems no problem?

Review: v_t of AdamW: $v_t = (1 - \beta_2) \sum_{\tau=1}^t \beta_2^{t-\tau} g_\tau^2$

- v_t becomes extremely large during next hundreds of steps
- $\alpha_t \frac{m_t}{\sqrt{v_t + \epsilon}} \rightarrow 0$, which means the optimization stops abnormally.

So when the gradient is too large, we need to scale it to a small enough level.

2. class Gradient_Clipper

Suppose all parameters of a model come from 5 Linear_Transform (5D to 5D), then the model has 5 parameter tensors (all 5*5 shape).

After one round of backpropagation, suppose gradient of all 5 parameters are equal to:

```
tensor([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

Let the parameter tensors be p_1, p_2, p_3, p_4, p_5 . $\|\nabla p_i\|_2 = \sqrt{25 * 1} = 5$

So the grad norm is $\sqrt{\sum_{i=1}^5 \|\nabla p_i\|_2^2} = \sqrt{125} = 11.18$

2. class Gradient_Clipper

- Assuming the acceptable upper limit is $\text{max_norm} = 0.01$:
- All parameters must be multiplied by the scaling factor $\frac{\text{max_norm}}{g+\epsilon}$ to ensure the total L2 norm size of gradients is reasonable.
- After clipping, the gradients of each parameter are $11.18/0.01$ times smaller:

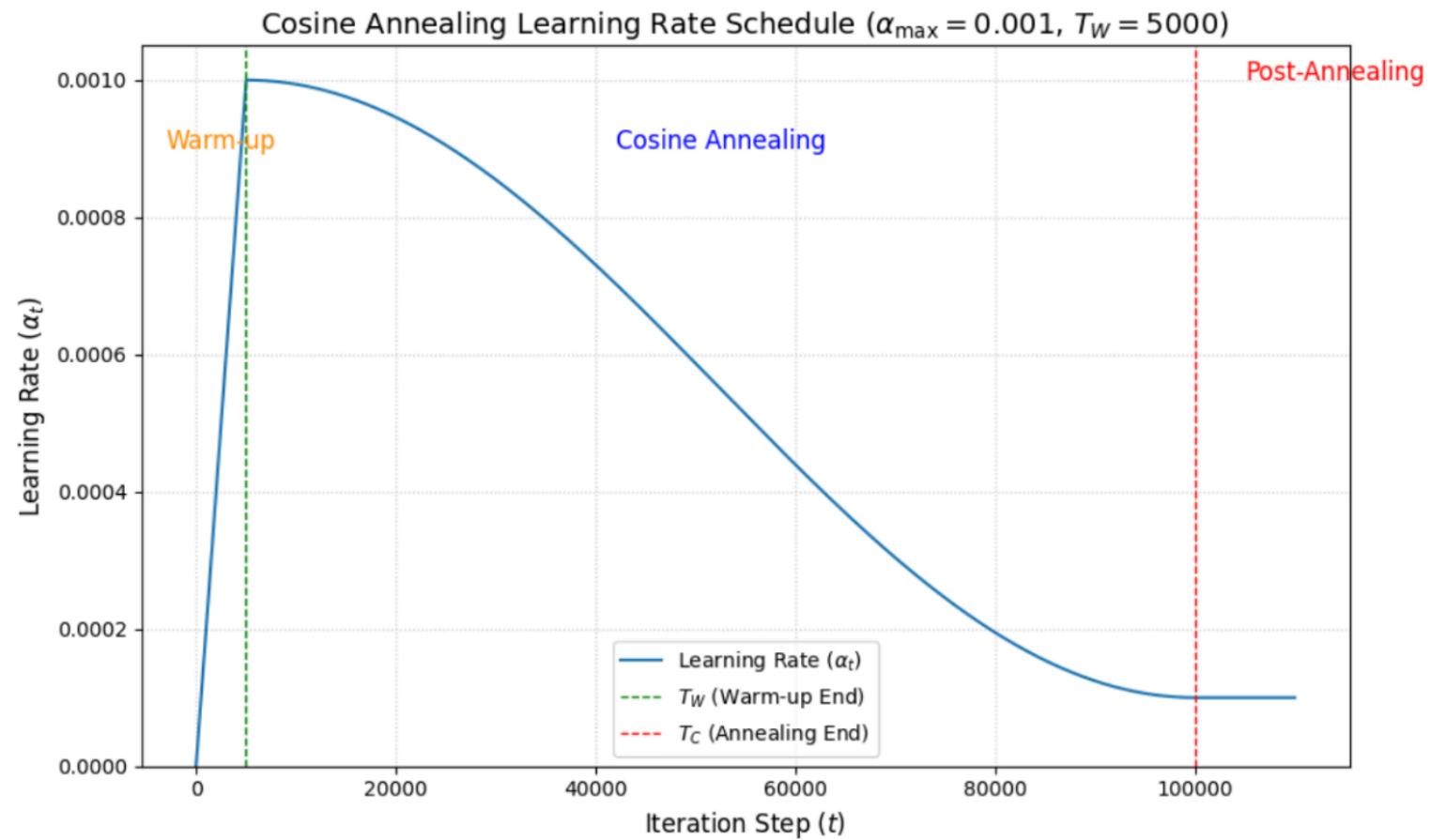
```
tensor([[0.0009, 0.0009, 0.0009, 0.0009, 0.0009],  
       [0.0009, 0.0009, 0.0009, 0.0009, 0.0009],  
       [0.0009, 0.0009, 0.0009, 0.0009, 0.0009],  
       [0.0009, 0.0009, 0.0009, 0.0009, 0.0009],  
       [0.0009, 0.0009, 0.0009, 0.0009, 0.0009]])
```

2. class Gradient_Clipper

```
class Gradient_Clipper:
    def __init__(self,max_norm:float):
        self.max_norm=max_norm
    def clip(self,parameters):
        total_norm=torch.sqrt(sum(p.grad.data.norm(2)**2 for p in parameters if p.grad is not None))
        for p in parameters:
            if p.grad is None:
                continue
            if total_norm<=self.max_norm:
                continue
            clip_factor=self.max_norm/(total_norm+1e-6)
            p.grad.data=p.grad.data*clip_factor
```

3. class Learning_Rate_Scheduler

Throughout the model training process, the learning rate is not constant:



3. class Learning_Rate_Scheduler

```
class Learning_Rate_Scheduler:  
    def __init__(self):  
        pass  
    def get_lr(self,step,lr_max,lr_min,Tw,Tc)->float:  
        if step<Tw:  
            lr=lr_max*step/Tw  
        elif step>Tc:  
            lr=lr_min  
        else:  
            lr=lr_min+0.5*(1+math.cos(math.pi*(step-Tw)/(Tc-Tw) ))*(lr_max-lr_min)  
    return lr
```

PART5: Checkpoints

A hidden problem

Usually we need hours even days to train a large model.....

Can we ensure there's nothing wrong during the whole process?

- For example, the computing power resource you applied for has been preempted, and everything crashed immediately.

To reduce the problem, we try to save to model every once in a while.

This is `checkpoint`.

What to save and load?

- Model parameters
- Optimizer parameters (such as m_t and v_t)
- Step now

We need to get them first, and try to save them as files in the disk.

1. How to Get All Parameters?

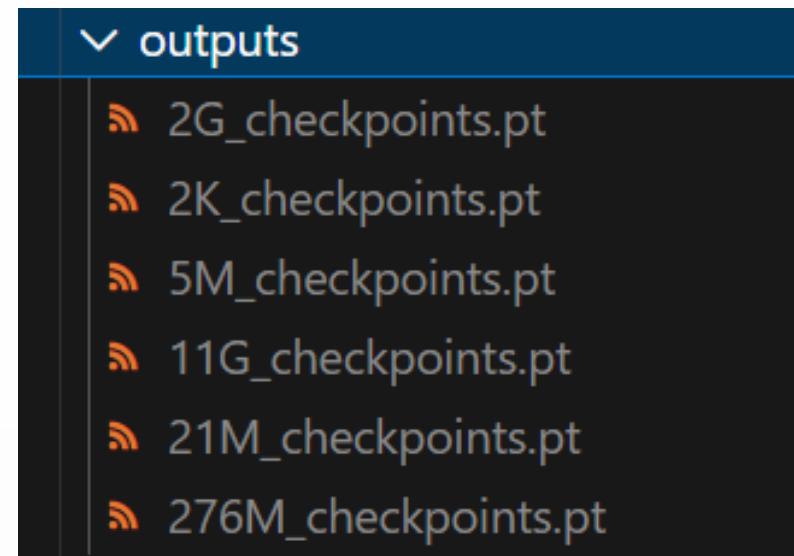
- PyTorch has a built-in `state_dict()` function that stores all parameters of the PyTorch module in dictionary form.

```
lm = Transformer_LM(  
    d_model=512,  
    num_heads=8,  
    d_ff=2048,  
    vocab_size=10000,  
    num_layers=2,  
    max_seq_length=128,  
    theta=10000,  
    dtype=torch.float32,  
    device="cpu"  
)  
  
states = lm.state_dict()  
for state_key in states:  
    print(state_key, states[state_key].shape)
```

```
transformer_blocks.0.RMSNorm_Attn.g torch.Size([512])  
transformer_blocks.0.RMSNorm_FF.g torch.Size([512])  
transformer_blocks.0.Multihead_Attn.q_proj.linear_matrix torch.Size([512, 512])  
transformer_blocks.0.Multihead_Attn.k_proj.linear_matrix torch.Size([512, 512])  
transformer_blocks.0.Multihead_Attn.v_proj.linear_matrix torch.Size([512, 512])  
transformer_blocks.0.Multihead_Attn.o_proj.linear_matrix torch.Size([512, 512])  
transformer_blocks.0.Multihead_Attn.rope.cos_values torch.Size([1, 128, 32])  
transformer_blocks.0.Multihead_Attn.rope.sin_values torch.Size([1, 128, 32])  
transformer_blocks.0.Feed_Forward.linear_w1.linear_matrix torch.Size([512, 2048])  
transformer_blocks.0.Feed_Forward.linear_w3.linear_matrix torch.Size([512, 2048])  
transformer_blocks.0.Feed_Forward.linear_w2.linear_matrix torch.Size([2048, 512])  
transformer_blocks.1.RMSNorm_Attn.g torch.Size([512])  
transformer_blocks.1.RMSNorm_FF.g torch.Size([512])  
transformer_blocks.1.Multihead_Attn.q_proj.linear_matrix torch.Size([512, 512])  
transformer_blocks.1.Multihead_Attn.k_proj.linear_matrix torch.Size([512, 512])  
transformer_blocks.1.Multihead_Attn.v_proj.linear_matrix torch.Size([512, 512])  
transformer_blocks.1.Multihead_Attn.o_proj.linear_matrix torch.Size([512, 512])  
transformer_blocks.1.Multihead_Attn.rope.cos_values torch.Size([1, 128, 32])  
transformer_blocks.1.Multihead_Attn.rope.sin_values torch.Size([1, 128, 32])  
transformer_blocks.1.Feed_Forward.linear_w1.linear_matrix torch.Size([512, 2048])  
transformer_blocks.1.Feed_Forward.linear_w3.linear_matrix torch.Size([512, 2048])  
transformer_blocks.1.Feed_Forward.linear_w2.linear_matrix torch.Size([2048, 512])  
final_norm.g torch.Size([512])  
final_layer.linear_matrix torch.Size([512, 10000])
```

2. How to Load All Parameters?

- PyTorch has a built-in `load_state_dict()` function that loads the values of all parameters of the PyTorch module from a state_dict dictionary.
- Prerequisite: Both must match completely!



3. class Checkpoint_Manager

- Objects to store: Model parameters, optimizer parameters, current iteration step.

```
class Checkpoint_Manager:  
    def __init__(self):  
        pass  
  
    def save(self, model, optimizer, iteration, save_path):  
        import os  
        os.makedirs(os.path.dirname(save_path), exist_ok=True)  
        state_model=model.state_dict()  
        state_optimizer=optimizer.state_dict()  
        checkpoint={  
            "model":state_model,  
            "optimizer":state_optimizer,  
            "iteration":iteration  
        }  
        torch.save(checkpoint,save_path)
```

3. class Checkpoint_Manager

- Objects to load: Model parameters, optimizer parameters, current iteration step.

```
def load(self,src_path,model,optimizer=None):
    checkpoint=torch.load(src_path)
    state_model=checkpoint["model"]
    if optimizer is not None:
        print(f"optimizer is not none")
        state_optimizer=checkpoint["optimizer"]
    iteration=checkpoint["iteration"]

    model.load_state_dict(state_model)
    if optimizer is not None:
        optimizer.load_state_dict(state_optimizer)
    return iteration
```

PART6: Start of Training: Input Parameter Decoding

To This Point...

We have sorted out:

- The architecture and parameters to pass of each model module.
- The implementation and parameters to pass of each training module.

Regarding passing parameters...

How do we specify the specific values of these parameters, **effectively** modify and pass certain values to each module?

1. Starting Training Scripts

- We can directly assign values to various parameters in the training script.
- When the training script is very complex, various functional modules will be ambiguous.
- We usually use a `bash` script to carry the required parameter values and start the execution of the `python` training script.

```
🐍 run_clm.py  
$ run_clm.sh
```

1. Starting Training Scripts

```
#!/bin/bash
python /home/kuangph/CS336-Assignment1/cs336_basics/run_clm.py \
--d_model 512 \
--num_heads 8 \
--d_ff 1344 \
--vocab_size 32000 \
--num_layers 8 \
--max_seq_length 256 \
--seq_length 256 \
--batch_size 48 \
--theta 100000 \
--device cuda \
--num_epochs 5.5 \
--lr 1e-4 \
--lr_min 1e-5 \
--warmup_ratio 0.05 \
--warmfix_ratio 0.9 \
--chunk_size 500000 \
--vocab_path /home/kuangph/CS336-Assignment1/data/vocab_32000.txt \
--merges_path /home/kuangph/CS336-Assignment1/data/merges_32000.txt \
--special_tokens "<|endoftext|>" \
--corpus_size "2G" \
--log_interval 20 \
--save_interval 500 \
--weight_decay 0.01 \
--betas 0.9 0.95 \
--eps 1e-8 \
--max_norm 1.0
```

- It is equal to enter a long command line in the terminal, containing information we need
- For `run_clm.py`, we need to properly decode information of command line.

2. Parameter Decoding

```
def parse_bash_args():
    parser=argparse.ArgumentParser()

    parser.add_argument("--d_model",type=int,default=512)
    parser.add_argument("--num_heads",type=int,default=8)
    parser.add_argument("--d_ff",type=int,default=1344)
    parser.add_argument("--vocab_size",type=int,default=32000)
    parser.add_argument("--num_layers",type=int,default=6)
    parser.add_argument("--max_seq_length",type=int,default=512)
    parser.add_argument("--seq_length",type=int,default=256)
    parser.add_argument("--batch_size",type=int,default=32)
    parser.add_argument("--theta",type=int,default=100000)
    parser.add_argument("--device",type=str,default="cuda")

    parser.add_argument("--num_epochs",type=float,default=10)
    parser.add_argument("--lr",type=float,default=1e-4)
    parser.add_argument("--lr_min",type=float,default=1e-5)
    parser.add_argument("--warmup_ratio",type=float,default=0.1)
    parser.add_argument("--warmfix_ratio",type=float,default=0.9)
    parser.add_argument("--corpus_size",type=str)
```

```
parser.add_argument("--chunk_size",type=int,default=500000)
parser.add_argument("--vocab_path",type=str,
                   default="data/vocab_32000.txt")
parser.add_argument("--merges_path",type=str,
                   default="data/merges_32000.txt")
parser.add_argument("--special_tokens",
                   type=str, nargs="*", default=["<|endoftext|>"])

parser.add_argument("--log_interval",type=int)
parser.add_argument("--save_interval",type=int)

parser.add_argument("--weight_decay",type=float,default=0.01)
parser.add_argument("--betas",type=float, nargs="*",
                   default=(0.9,0.95))
parser.add_argument("--eps",type=float,default=1e-8)

parser.add_argument("--max_norm",type=float,default=1.0)

args=parser.parse_args()
return args
```

2. Parameter Decoding

```
args=parse_bash_args()  
  
d_model=args.d_model  
num_heads=args.num_heads  
d_ff=args.d_ff  
vocab_size=args.vocab_size  
num_layers=args.num_layers  
max_seq_length=args.max_seq_length  
seq_length=args.seq_length  
batch_size=args.batch_size  
theta=args.theta  
dtype=torch.float32  
device=args.device
```

```
num_epochs=args.num_epochs  
lr_max=args.lr  
lr_min=args.lr_min  
warmup_ratio=args.warmup_ratio  
warmfix_ratio=args.warmfix_ratio  
  
chunk_size=args.chunk_size  
vocab_path=args.vocab_path  
merge_path=args.merges_path  
special_tokens=args.special_tokens  
corpus_size=args.corpus_size
```

PART7: Complete Script

Review of Model Training Process

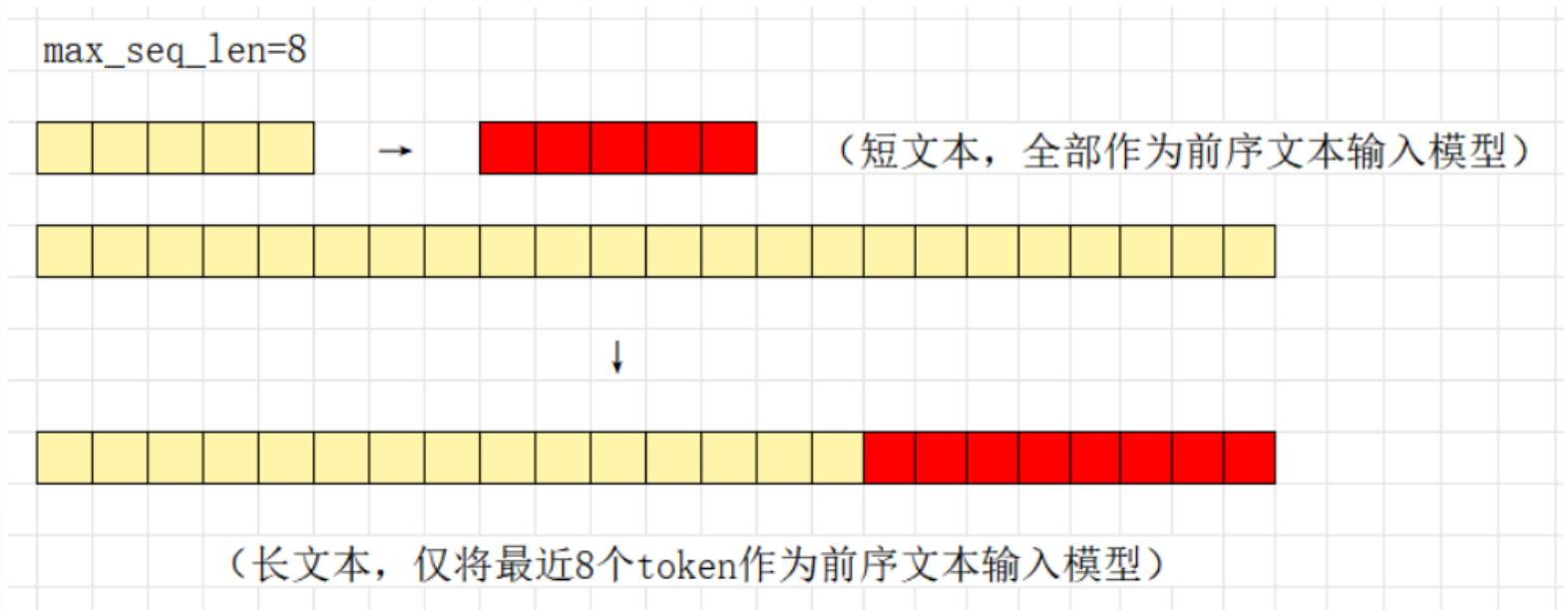
- Receive parameters, establish model and training modules.
- Determine the total number of iterative optimization steps.
- For each step:
 - Sample data.
 - Forward.
 - Calculate loss.
 - Calculate gradients and clip.
 - Schedule the learning rate.
 - Update parameters.
- It is recommended to output real-time status during the process, such as the current loss function value, current text prediction situation, etc.

Code Explanation

PART8: Autoregressive Decoding

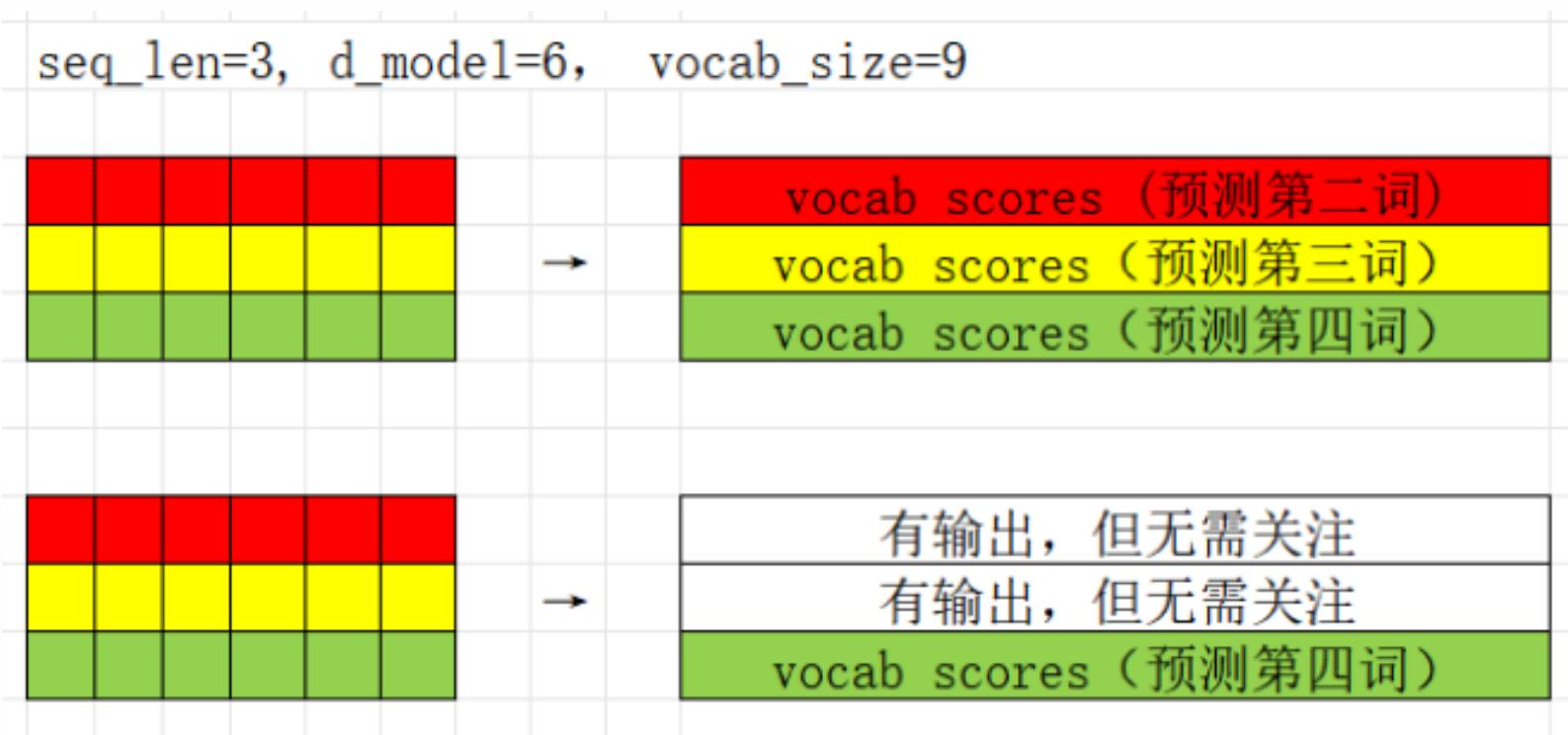
Determining Input Text

During training, fixed-length text is often sampled as input to the model. However, the total length of decoded text may be unpredictable.



Sampling Output Values

- During training, we need to know the output at all positions to calculate the loss value;
- During decoding, we only care about "what is the next word"!



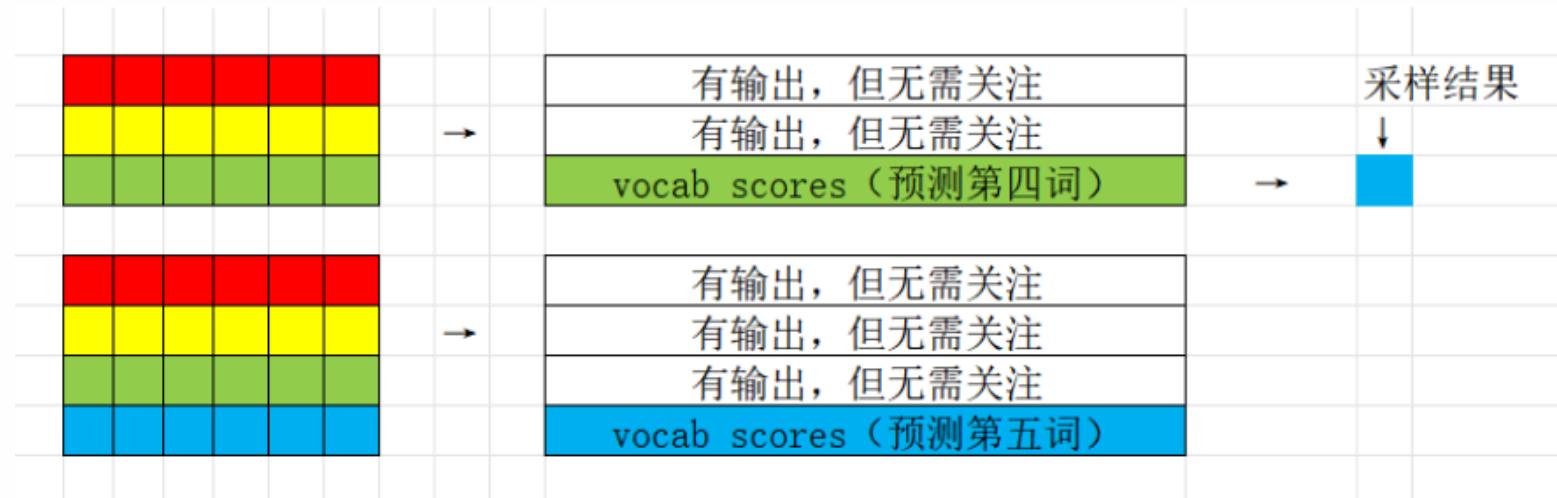
From Vocab Scores to Token Sampling

After receiving vocab scores:

- Repetition penalty: Reduce scores of recently appeared tokens to avoid repeating the same word.
- Temperature sampling: Scale all scores by the same factor to make softmax differences larger/smaller.
- Softmax: Convert scores into a probability distribution.
- Randomly sample the next token according to the probability distribution.

Autoregressive

The sampled output this step is a part of input next step.



Code Explanation and Practical Demonstration